Hide and Seek: Exploiting and Hardening Leakage-Resilient Code Randomization*

Robert Rudd
MIT Lincoln Laboratory

Thomas Hobson
MIT Lincoln Laboratory

Richard Skowyra
MIT Lincoln Laboratory

David Bigelow
MIT Lincoln Laboratory

Veer Dedhia
MIT Lincoln Laboratory

Stephen Crane
University of California, Irvine

Christopher Liebchen
TU Darmstadt

Per Larsen
University of California, Irvine

Lucas Davi
TU Darmstadt

Michael Franz
University of California, Irvine

Ahmad-Reza Sadeghi
TU Darmstadt

Hamed Okhravi
MIT Lincoln Laboratory

Abstract

Information leakage vulnerabilities can allow adversaries to bypass mitigations based on code randomization. This discovery motivates numerous techniques that diminish direct and indirect information leakage: (i) execute-only permissions on memory accesses, (ii) code pointer hiding (*e.g.*, indirection or encryption), and (iii) decoys (*e.g.*, booby traps). Among the proposed leakage-resilient defenses, Readactor is the most comprehensive solution that combines all these techniques. In this paper, we conduct a systematic analysis of recently proposed execute-only randomization solutions including Readactor, and demonstrate a new class of attacks that bypasses them generically, highlighting their limitations. We analyze the prevalence of opportunities for such attacks in popular code bases and build three real-world exploits to demonstrate their practicality. We then implement and evaluate a new defense against our attacks. Our evaluation shows that our new technique is practical and adds little additional performance overhead (9.7% vs. 6.4%).

1 Introduction

Return-oriented programming (ROP) [53] emerged in response to the widespread adoption of defenses such as W⊕X (Write⊕eXecute) [46]. Despite numerous advances in this domain, comprehensively protecting native code written in C/C++ from ROP and other code-reuse attacks remains an open area of research [58].

Two classes of code-reuse defenses have been studied in the literature, one based on enforcement [1, 37, 44] and the other based on randomization [8, 38]. In this paper, we focus on randomization-based defenses. These defenses have the advantage of being efficient and scalable to complex software such as browsers but are less effective in the presence of information leakage vulnerabilities [52, 57] that reveal the effects of randomization. Direct leakage of memory content (a.k.a., memory disclosure) [55, 57], indirect leakage of addresses from the stack or heap [21], and remote side-channel attacks [51] are different forms of information leakage that have been used successfully to bypass recent randomization-based defenses [16, 21, 24].

In response to information-leakage attacks, researchers have used execute-only memory (X-only) to build leakage-resilient, randomization-based defenses [5, 19, 27]. These schemes deploy some or all of the following techniques: they a) enforce execute-only permissions on code pages to mitigate direct information leakage, b) introduce an encryption or indirection layer (e.g. by routing calls and returns through *trampolines*) to prevent code pointers from indirectly leaking the code layout, and finally c) they randomize the trampoline layout to create uncertainty for an attacker. Defenses such as XnR [5] and HideM [27] mitigate direct code leakage, but remain vulnerable to indirect leakage and remote side-channel attacks. ASLR-Guard [41] prevents indirect leakage through code pointers but remains vulnerable to direct leakage. So far, the most comprehensive X-only defense is Readactor [19], which combines all of the above mentioned techniques. A recent extension to Readactor, called Readactor++ [20] additionally mitigates whole-function reuse attacks such as COOP attacks that inject counterfeit objects into memory [50] and detects brute-force guessing attacks by inserting decoy trampolines also known as booby traps [18].

Goals and Contributions. In this paper, we systematically investigate the effectiveness of randomization-based schemes that deploy execute-only and code pointer hiding techniques. We demonstrate a new class of code-reuse attacks which we call Indirect Code Pointer-Oriented Programming (ICPOP)[1] that can generically bypass execute-only defenses. The intuition behind ICPOP is that execute-only permissions apply just to

*This work is sponsored by the Department of Defense under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

[1] Pronounced "Icy Pop"

code, not code pointers (e.g., function pointers and return addresses). Code pointers must be readable for programs to function correctly. Various execute-only defenses use some form of hiding (indirection or encryption) to protect these code pointers, but these alternative code pointer representations remain useful to an attacker.

We demonstrate that an attacker can *profile* this layer of indirection in code pointers by observing the state of the protected program remotely, and extract these indirect code pointers. We then show that solely by reusing these indirect code pointers, an attacker can achieve malicious behavior without explicitly requiring read access to the diversified code. We call our attack Indirect Code Pointer-Oriented Programming (ICPOP) because it is a type of code reuse, but at the abstraction of indirect code pointers.

In order to accurately profile indirect code pointers in a running process remotely, we devise a new attack technique which we call Malicious Thread Blocking (MTB). To chain indirect code pointers, we generalize the COOP technique [50] to imperative programming languages lacking object orientation, in what we call Counterfeit Procedural Programming (CPP).

Using these techniques, we build two real-world ICPOP exploits against Nginx and one against Apache to hijack the control flow in the presence of full-featured Readactor. These attacks are not limited to Readactor; we discuss the generality of these attacks against other recent defenses and show that many of them are also vulnerable to ICPOP.

Using the lessons learned from these attacks, we build and evaluate a countermeasure against them. We create a technique called Code Pointer Authentication that prevents ICPOP by establishing a form an authentication over the layer of indirection used to protect the code pointers. We directly augment the Readactor system[2] to demonstrate the feasibility and effectiveness of our technique.

Finally, we discuss and mitigate some implementation flaws in modern X-only defenses such as using memory access channels not mediated by page permissions (e.g., malicious software-based direct memory access (DMA)).

In summary, our contributions are as follows:

- We present a new class of attacks that can generically bypass X-only defenses. We build three real-world exploits against Nginx and Apache.
- We present two techniques to accurately profile (Malicious Thread Blocking) and chain (Counterfeit Procedural Programming) ICPOP gadgets.
- We present and discuss some implementation challenges in modern X-only defenses.
- We propose and implement a countermeasure against our attacks, and evaluate its performance and effectiveness.
- We discuss the generality of our attacks against other recent leakage-resilient defenses.

[2]Crane et al. [19] kindly shared their implementation with us.

2 Threat Model

Our threat model assumes that a remote attacker uses a memory corruption vulnerability to access arbitrary memory and achieve remote code execution on the victim machine. We assume W$\oplus$X is deployed to prevent code injection and modification. Moreover, we assume that the software executing on the target system is protected by a leakage-resilient randomization-based defense capable of stopping conventional code reuse [15, 53] and just-in-time code-reuse attacks [55]. In particular, we assume that the target system:

1. maps code pages with execute-only permissions to prevent direct leakage [5, 19, 27],
2. hides, encrypts or obfuscates code pointers to prevent indirect leakage [19, 41],
3. randomizes the code layout at any granularity up to individual instructions [19, 31, 35, 47], and
4. randomizes the entries of function tables [20] rendering whole-function reuse attacks cumbersome [50].

Our threat model is consistent with related work on leakage-resilient randomization-based defenses against code reuse.

While strong enforcement-based and randomization-based defenses in the literature have assumed that the adversary can read and write arbitrary memory (modulo page permissions), we demonstrate that practical attacks can in fact be mounted by a less powerful adversary.

3 Indirect Code Pointer-Oriented Programming (ICPOP) Attack

3.1 Overview

Current state-of-the-art randomization-based defenses [5, 6, 12, 19, 20, 27, 41] aim to prevent code-reuse attacks by limiting an attacker's ability to disclose the code layout, either by leaking the code itself or by leaking code pointers. As noted in Section 2, the adversary is assumed to have arbitrary read and write capabilities. Two primary techniques are employed to stop these adversaries:

- *Execute-only permissions* prevent read accesses to code pages (existing W$\oplus$X policies already prevent writes to code pages). Thus, any attempts by an attacker to directly disclose the locations and contents of code pages will lead to a segmentation violation.
- *Code pointer hiding* seeks to prevent indirect memory disclosure by changing how pointers to code are

stored in attacker-observable memory. Some approaches alter the pointer representation using fast XOR encryption [12, 17, 41]. Others use indirection mechanisms [6, 19]. For instance, Readactor replaces all observable code pointers with pointers to trampolines. A forward trampoline is simply a direct jump to a function stored in execute-only memory. Because the location of the forward trampoline and the function it jumps to are randomized independently, attackers cannot infer the function layout by observing the trampoline layout.

In this section, we describe a code-reuse attack that generically circumvents execute-only memory defenses, even under the strong assumption that execute-only permissions are universally enforced. In strong execute-only defenses, in addition to additional page permissions, encryption or indirection is used to protect code pointers during execution. We show how an attacker can indeed use these indirect code pointers to launch meaningful exploits. This is achieved by profiling the indirect code pointers to determine the underlying original code to which they point. We demonstrate how multiple profiled indirect code pointers can be used together to launch a chained attack (ICPOP) akin to traditional ROP, but at the granularity of code segments pointed to by these indirect code pointers.

3.2 Profiling & Malicious Thread Blocking

The goal of profiling is to determine the original function `F()` that is invoked by an indirect code pointer `icptr`. Various X-only defenses use different names for these indirect code pointers. For example, Readactor [19] calls them trampoline pointers, while ASLR-Guard [41] calls them encryption code locators. We use the generic name "indirect code pointers", but the discussions apply to these and similar defenses.

An attacker who can identify the mapping of `icptr`→`F` can redirect control flow to `F` in an indirect way via `icptr`. "→" denotes that `icprt` is the pointer to the indirection layer (trampoline or encrypted pointer) that corresponds to function `F`.

To infer this mapping, we exploit the fact that programs execute in a manner that inherently leaks information about the state of execution. Knowledge about the execution state of a program at the time of a memory disclosure enables us to infer the `icptr`→`F` mapping from a leaked `icptr`.

An attacker can use her knowledge about function addresses in the unprotected version of the program to infer the locations of indirect code pointers in the protected version. We illustrate this idea in Figure 1. By observing what functions pointers are placed in observable memory (*i.e.*, stack or heap) in the unprotected version, an attacker can infer that the pointers observed in the protected version must be the corresponding indirection pointers of the same functions.

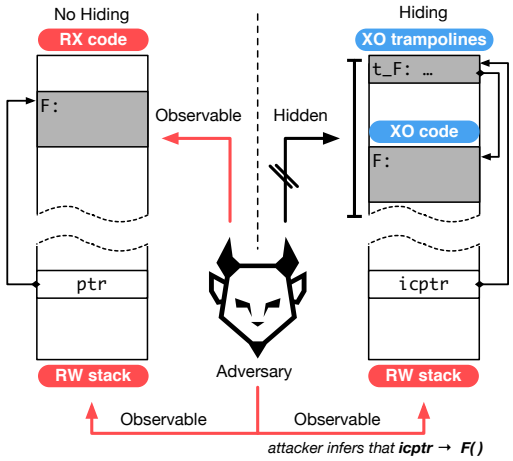

Figure 1: Profiling of Indirect Code Pointers

At a high-level, to perform the profiling, the attacker collects a list of function pointers from an unprotected version of the application *offline*, then she collects some indirect code pointers from the protected application in an *online* manner by sending the victim a few queries and observing parts of its data memory (e.g., stack). This allows the attacker to create a mapping between the discovered indirect code pointers and their underlying functions. The attacker can then chain these indirect code pointers to achieve the desired malicious behavior. Since the code snippets pointed to by these indirect code pointers behave like traditional ROP gadgets we call them ICPOP gadgets. Although these steps sound straightforward, in practice the attack faces a number of technical challenges. Here, we describe the techniques we devised to overcome these challenges.

A naïve approach to create an accurate mapping of indirect code pointers to underlying functions can rely on repeated stack disclosures and precise timing of the leakage. However, since the state of the system changes very rapidly, this can result in inaccuracies in the mappings which eventually causes a crash at exploitation time. To enhance the precision of the mapping, we devised a technique which we call Malicious Thread Blocking (MTB).

In the case of programs that utilize threading, we can employ MTB to enable us to profile a broader range of indirect code pointers and avoid dependence upon strict timing requirements for triggering the disclosure vulnerability.

The approach of MTB is to use one thread, T_A, to cause another thread, T_B, to hang at an opportunistic moment by manipulating variables that cause T_B's execution to block, *e.g.*, by maliciously locking a mutex. By oppor-

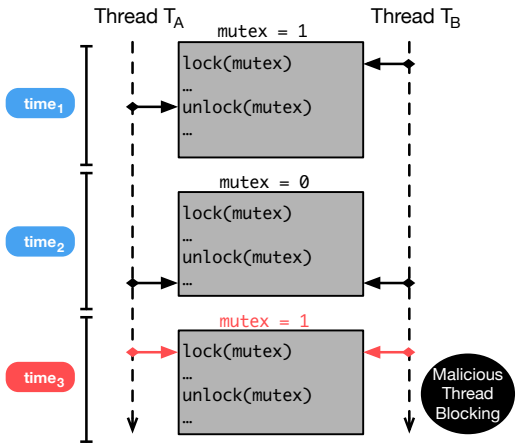

Figure 2: Normal vs. Malicious Thread Blocking

tunistically blocking a thread, we can more easily locate and map desired trampoline pointers without worrying about rapid changes in memory. A memory disclosure vulnerability may be triggered in T_A that enables memory inspection at a known point in execution in T_B. Note that this technique gets around any timing unpredictability that the attacker may face when trying to trigger a disclosure in thread T_A at the appropriate time in execution for thread T_B. The idea of this approach is illustrated in Figure 2.

As one example of this technique in practice, we show in Section 4 how an attacker can lock a mutex in Nginx to cause a thread to block upon returning from a system call. Triggering a memory disclosure vulnerability in another thread at any point after the system call enables the attacker to inspect a memory state that she knows contains trampoline pointers relevant to the system call. To more easily distinguish one system call from another, the attacker can supply a unique input and scan disclosed memory for that input.

3.3 Passing Proper Arguments

After the attacker has mapped relevant indirect code pointers to their underlying functions, it is straightforward to redirect control flow to one of the functions. For the purpose of control flow hijacking, knowing an indirect code pointer address is just as good as knowing the address of a function.

Consider the following code fragment:

```
call(int arg1, int arg2) {
    fptr(arg1, arg2); }

call_with_defaults() {
    fptr(default_arg1, default_arg2);  }
```

If the attacker modifies the region of memory containing `fptr`, the next invocation of `call` or `call_with_defaults` will be redirected to an indirect code pointer chosen by the attacker. Unlike ROP and similar attacks, this redirection is consistent with the high-level semantics of C, thus unaffected by any underlying randomization (instruction-level, basic block-level, function-level, or library-level). In a valid C program `fptr` can potentially point to any function in the program.

Hijacking control flow in this manner does have limitations. If the attacker ends up hijacking a call like the one in `call`, the attacker will have very limited ability to control the arguments. The x86_64 ABI mandates a calling convention in which the first few arguments must be passed via register. It is much more difficult to control a value in a register than it is to control a value in memory. Some diversity techniques further complicate this by randomizing how registers are allocated to variables and how registers are saved to and restored from the stack [19, 47].

An attacker can overcome these defenses by concentrating on hijacking calls like the call in `call_with_defaults`. `call_with_defaults` invokes `fptr` on global variables. As global variables are stored in memory, they are trivial to modify. If an attacker is able to locate a function like `call_with_defaults`, she will be able to redirect control to a function of her choosing, with up to two arguments of her choosing.

In our experiment with Nginx and Apache, we observed many such opportunities as we will discuss in our real-world exploits in Section 4.

3.4 Chaining via Counterfeit Procedural Programming

An attacker wishing to chain multiple ICPOP gadgets together faces another challenge: after calling an indirect code pointer, the execution returns to the original call site. This makes it difficult for the attacker to take the execution control back after a single function call. For example, in Readactor, trampolines consist of a call immediately followed by a jump to the original call site; any redirected call will end with a return to normal program execution. Theoretically, there is a window, potentially very narrow, between the invocation of a redirected call and the return of the redirected call in which an attacker may modify the return address to maintain control. This approach requires a very precise level of timing which may be difficult to achieve in practice.

To overcome this, we introduce a technique we call Counterfeit Procedural Programming (CPP), which only depends on the semantics of language-level abstractions, similar to COOP [50], but which does not depend on a dynamic dispatch implementation based on vtables.

In a COOP attack, an attacker chains a set of virtual function gadgets (vfgadgets) together using main-

```
while (task) {
    task->fptr(task->arg);
    task = task->next;
}
```

Figure 3: A loop with a corruptible call site

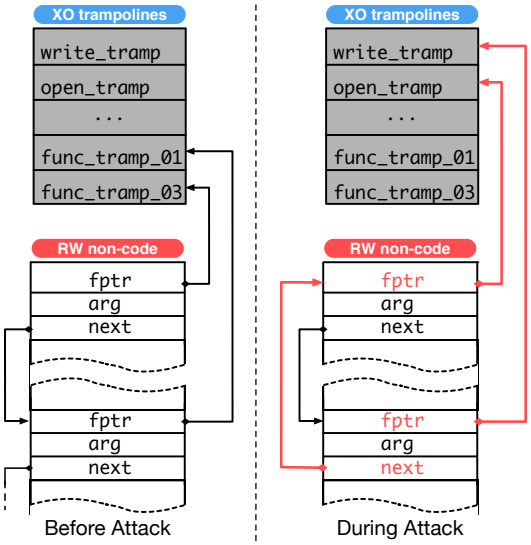

Figure 4: Counterfeit Procedural Programming

loop gadget (ML-G). The attack abuses language-level semantics widely found in C++ applications (loops over virtual functions).

The CPP attack is the generalization of COOP to non-object-oriented programming languages such as C. In a CPP attack, an attacker chains a set of indirect code pointers (i.e., ICPOP gadgets) using loops that contain indirect call sites in their body analogous to a main-loop gadget. The attack abuses language-level semantics widely found in C applications (loops over function pointers).

An appropriate main-loop gadget in CPP is a loop that:

1. has a loop condition that can be subverted by an attacker (e.g., the loop condition is in RW memory)
2. has code pointers in its body

A simple example is presented in Figure 3. If `task` points to attacker controllable memory, the attacker can cause the program to perform calls to multiple functions of her choosing by creating several counterfeit `task` structures and setting up their `task->next` pointers to point to the next ICPOP gadget. When the loop runs, ICPOP gadgets are executed one by one without loss of control on attacker's side. We depict this attack graphically in Figure 4.

While some defenses implement register randomization to prevent chaining computations together, it does not prove to be an effective deterrent in this situation. The high-level semantics of the call dictate that the first argument will be taken from `task->arg`. Thus, any randomization technique that preserves the x86_64 ABI will generate code which moves the value located at `task->arg` into `RDI`. As a result, our method of chaining ICPOP gadgets using CPP succeeds regardless of any underlying randomization that might be deployed.

4 Real-World Exploits

In this section we present three real-world exploits combining various techniques described earlier. The first two exploits target Nginx and the third targets the Apache HTTP Server. The attacks are tested on Readactor as a proof-of-concept, but they are generally applicable to other execute-only defenses as we discuss in Section 9. Nginx attack 1 uses profiling to locate call trampolines for `open` and `_IO_new_file_overflow` and uses these to hijack control. Nginx attack 2 and the Apache attack use profiling to locate call trampolines for functions that eventually reach `exec`.

4.1 Nginx Attack 1

Our setup consists of Nginx 1.9.4 configured with support for thread-based asynchronous I/O.

The aim of our attack is to cause Nginx to perform a malicious write to a file from a buffer located in execute-only memory. This requires locating addresses of functions that open and write files. We must also locate an indirect call site with enough corruptible arguments to call our target functions.

We began by inspecting the Nginx source code for suitable corruptible call sites. We were able to find an indirect call site which retrieved both of its arguments from memory in Nginx's main loop for worker threads. On line 335 of `core/ngx_thread_pool.c`, the following call is made:

```
task->handler(task->ctx, tp->log);
```

This call site is ideal for our purposes: both the function pointer itself and the arguments are obtained by referencing a field of a struct retrieved from memory, and are thus corruptible.

While this call site is suitable for calling `open`, which only requires two arguments, it does not allow us to call `write`, which requires three. As it seemed unlikely that we would find a better callsite, we began searching for ways to perform a write via a function that only takes two arguments. We eventually found `_IO_new_file_overflow`, an internal function in the GNU C Library (glibc) used when a write to a file is about to overflow its internal buffer. The signature for this function is included below:

```
_IO_new_file_overflow(_IO_FILE *f,int ch)
```

`f` is a pointer to an `_IO_FILE`, glibc's internal version of the C standard library type `FILE`. `ch` is the character that was being written when the overflow occurred. If a pointer to an attacker controlled `_IO_FILE` were to be

passed to this function, they would be able to reliably perform a write from an arbitrary buffer of arbitrary size to an arbitrary file descriptor.

To locate the indirect code pointers of these functions we perform profiling as described in Section 3. `_IO_new_file_overflow` can be located using only analysis of in-memory values. Locating `open`, however, requires the use of the MTB technique. At a high level this attack proceeds in four phases:

1. Locate a mutex for MTB
2. Profile a call trampoline for `open` (our first ICPOP gadget)
3. Profile a trampoline for `_IO_new_file_overflow` (our second ICPOP gadget)
4. Corrupt Nginx's task queue so that a worker thread makes calls to our profiled trampolines using the CPP technique

For the sake of brevity, we describe the details of this attack in Appendix A.

4.2 Nginx Attack 2

We now illustrate the generality of our techniques by performing a second attack against Nginx which both (1) targets different functions and (2) corrupts a different call site.

This attack relies on invoking Nginx's master process loop from an attacker-controlled worker in order to trigger a specific signal handler and cause arbitrary process execution. There are three phases to this attack:

1. Use profiling to get the address of the master process loop
2. Use MTB to corrupt a function pointer to point at the master process loop
3. Set global variables via MTB to cause the master process loop to call `exec` under attacker-chosen parameters

For the sake of brevity, we describe the details of this attack in Appendix B.

4.3 Apache Attack

Finally, we describe an attack using similar techniques against the Apache HTTP Server. While previous attacks have focused on Nginx, MTB and profiling are general and can be applied to other targets. Arbitrary process execution can be achieved on the Apache web server using a similar approach:

1. Use profiling to find the indirect code pointer of the `exec`-like function `ap_get_exec_line`
2. Use MTB to corrupt a function pointer to point at `ap_get_exec_line` and cause an `exec` call under attacker control

For the sake of brevity, we describe the details of this attack in Appendix C.

All exploits succeeded in control hijacking while Apache and Nginx were protected by full-featured Readactor.

5 X-Only Implementation Challenges

This section describes the imperfections of enforcing execute-only permissions in modern systems. We discuss two attacks that are widely available in x86 UNIX-based systems. Rather than indicating weaknesses in a particular execute-only technique, we highlight the intricacies of enforcing execute-only permissions universally.

5.1 Forged Direct Memory Access Attack

Execute-only defenses protect code pages from direct read accesses by applying additional permissions to memory pages in software [5] or hardware [19, 27]. This enforcement, however, applies only to regular memory accesses (*i.e.*, TLB-mediated). Accesses performed by devices capable of Direct Memory Access (DMA), *e.g.*, GPUs, disk drives, and network cards, do not undergo translation by the MMU and are unaffected by page permission. We call these accesses "non-TLB-mediated."

The idea of exploiting systems via DMA is well studied, especially in the context of DMA-capable interfaces with external connectors, *e.g.*, IEEE 1394 "Firewire" and Thunderbolt [49]. DMA attacks have been successfully used against systems in both physical and virtualized environments.

As described in the threat model (Section 2), we are mainly concerned about a remote attacker. For that, the attacker must be able to perform software-based DMA from a user space application. Typically, user space applications cannot directly make requests to DMA-capable devices. However, some user space functionality is implemented via the kernel requesting a device to perform a DMA against a user space controlled address. Examples of this include OpenCL's `CL_MEM_USE_HOST_PTR` flag and Linux's `O_DIRECT` flag.

An attacker can use Linux's `O_DIRECT` flag to maliciously request software-based DMA to bypass execute-only, thus alleviating the need for compromised peripheral devices or hardware attacks. We call such an attack a Forged DMA (FDMA) attack, and briefly demonstrate its feasibility. The novelty of FDMA is its broad applicability remotely and from user space applications. Unlike well-studied DMA attacks such the one used in bypassing Xen [61], FDMA does not require a malicious device or kernel permissions.

Applications that use the `O_DIRECT` flag natively are vulnerable to our FDMA attack. More surprisingly though, even applications that never use the `O_DIRECT` flag, but pass the flags to file read or write operations through the flags variable residing in data memory are also vulnerable to this attack. An attacker can perform a simple data-only attack to maliciously change the flags

variable to O_DIRECT in order to force a regular file operation to become a DMA access.

We investigated the prevalence of direct I/O and flags variables is in popular real-world software packages. Our analysis focused on Internet-facing web servers (AOLserver, Apache, Boa, lighttpd, Nginx, OpenSSH, Squid, and Firebird) due to their exposure and database managers (Hypertable, MariaDB, Memcached, MongoDB, MySQL, PostgreSQL, Redis, and SQLite) due to their focus on fast I/O. The results indicate that the majority of web servers and database managers (13 out of 16) do not natively use the O_DIRECT flag; however, 10 out of 16 of them (AOLserver, Nginx, OpenSSH, Squid, Firebird, Hypertable, MongoDB, MySQL, PostgreSQL, and SQLite) use variables to store flags that can be corrupted by an attacker to set the O_DIRECT flag. As such, an attacker can use an FDMA attack in these applications to read X-only code pages to build a traditional ROP attack even in the presence of X-only defenses. The FDMA attack would obviate the exploit, and does not require an ICPOP attack to bypass X-only.

5.2 Procfs Attack

The proc filesystem is another implementation challenge that can obviate X-only bypasses.

The proc filesystem is a file-like structure that contains information about each process. It is implemented for a variety of UNIX-like operating systems [26, 36]. In this paper, we focus on the Linux implementation of procfs [11].

The Linux kernel creates a directory for each process which can be accessed via /proc/<process id>/. Processes can access their own directory via /proc/self/. The files within the procfs directory are, for the most part, treated in the same way as any other file in a filesystem. They have ownership settings and assigned permissions, and are accessed via the same mechanisms as any other file. Through them, a wealth of information about the process is made available: details about program invocation, processing status, memory access, file descriptors, networking, and other internal details.

Several of the procfs files (e.g., auxv, maps, numa_maps, pagemaps, smaps, stat, syscall, exe, stack, and task) include memory addresses that reveal information about the randomized code layout. The mem file even allows direct disclosure of the process memory regardless of memory permissions.

To carry out a procfs attack, the attacker needs to (1) discover the location of a suitable piece of executable memory, and (2) leak executable memory directly by corrupting the filename argument to a file read operation. The maps and smaps files provide, among other things, the starting and ending addresses of each mapped memory region, along with that region's memory permissions and the file (if any) with which the region is associated. After that, reading the mem file directly leaks the executable regions. Note that even when the vulnerability does not allow arbitrary file reads, the procfs attack can be mounted by performing a data-only corruption on any file read operation.

The procfs attack also allows a leakage of the actual code pointers followed by a traditional ROP attack, without requiring the sophistication of an ICPOP attack.

6 Code Pointer Authentication

We carefully considered how to best mitigate the execute-only bypass presented in Section 3. The pointer harvesting technique can be prevented by isolating control data from non-control data [37]. The control-flow hijacking step can be mitigated by using control-flow integrity. These enforcement-based techniques come with their own set of challenges and weaknesses as demonstrated by numerous recent bypasses [13, 24, 25, 39]. Rather than swapping one set of challenges for another, we explore whether leakage-resilient diversity can be extended to mitigate ICPOP attacks.

Recall that code pointer hiding via trampolines already limits the set of addresses that are reachable from an attacker-controlled indirect branch. Even if an attacker discloses all trampoline pointers, only function entries, return sites, and individual instructions inside trampolines are exposed. We therefore implemented an extension to the Readactor code pointer hiding mechanism, which we call Code Pointer Authentication (CPA). CPA adds authentication after direct calls and before indirect calls to prevent the control-flow hijacking step explained in Section 3.3 and thus mitigate ICPOP attacks. One of the benefits of randomization-based defenses is that they do not rely on static program analysis which helps them scale to complex, real-world code bases. Without static program analysis, however, we must use different techniques to authenticate direct and indirect calls since we do not know the set of callees in the latter case.

6.1 Authenticating Direct Calls & Returns

Our general approach to authenticate direct calls uses cookies. A cookie is simply a randomly chosen value that is loaded into a register by the caller and read out and checked against an expected value by the callee. Similarly, the callee loads a separate cookie into a register before returning, and the register is checked for the expected value right after the return. Each function has two unique, random cookies: one to authenticate direct calls to the function (forward cookie, FC) and another to authenticate returns (return cookie, RC). Our prototype implementation chooses cookie values at compile time; a full-featured implementation would randomize the cookie values at load time so they vary between executions. Because the instructions that set and check cookies are stored in execute-only memory and the reg-

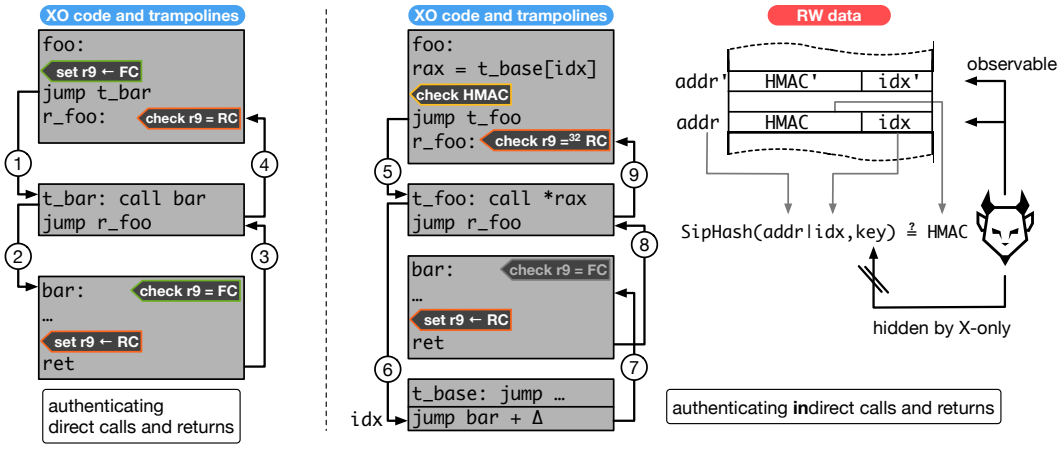

Figure 5: Code pointer authentication. Direct calls and returns are illustrated in the leftmost third of the figure; indirect calls and returns are shown in the rightmost two thirds. Light grey boxes contain execute-only code and white boxes contain data. Dark grey labels show where we insert additional instructions to prevent address harvesting attacks. The $=^{32}$ operator in the check after edge 9 indicates that we only check the lower 32-bits of the return cookie.

ister storing the cookie is cleared directly after the check, attackers cannot leak or forge the cookies.

The left-hand side of Figure 5 shows how we authenticate an example direct function call from `foo` to `bar`. Dark grey labels indicate how we extend the Readactor code pointer hiding technique with authentication cookies. Before transferring control to the direct call trampoline `t_bar` along control flow edge ①, we load `bar`'s forward cookie into a scratch register. Edge ② transfers control from `t_bar` to `bar`. The prologue of `bar` checks that the register contents match the expected forward cookie value and clears the register to prevent spilling its contents to memory. Before the `bar` function returns along edge ③, we load the backward cookie for bar into the same scratch register. At the return site in `foo`, we check that the register contains the backward cookie identifying `bar` as the callee. The return site then clears the register.

The return address pushed on the stack by the call instruction in `t_bar` leaks the location of the following jump instruction as well as the direct call itself. If the adversary manipulates an indirect branch to execute control flow edge ②, the check at the target address will cause the forward cookie check to fail and thus the attack to fail. Analogously, redirecting control to flow along edge ④ will cause the check at `r_foo` to fail.

6.2 Securing Indirect Calls & Returns

Without fine-grained static program analysis, we cannot, at compile-time, know the target of an indirect call and thus enforce bounds on the program control flow. Cookies, as we use in the direct call case, are therefore not applicable to indirect calls. However, we can still authenticate that the function pointer used in an indirect calls was correctly stored and not maliciously forged without requiring any static analysis.

All function pointers in a Readactor protected program are actually pointers to trampolines which obscure the true target address. Inspired by the techniques of CCFI [42], we change the representation of trampoline pointers (which are stored in attacker observable memory) to allow for authentication. In Readactor's code pointer hiding mechanism, a trampoline pointer is simply the address of the forward trampoline. With CPA, the trampoline pointer representation is composed of a 16-bit index (`idx`) into a table of trampolines (starting at `t_base`) and a 48-bit hash-based message authentication code, HMAC. We show two such pointers in the right-hand side of Figure 5. Using a trampoline index prevents leakage of the forward trampoline pointer address since the base address of the array of forward trampolines `t_base` can be hidden in execute-only code. We found that programs need less than 2^{16} forward pointers in practice, so it suffices to use the lower 16 bits of a 64-bit word for the index (this can be adjusted as needed for larger applications). We compute the HMAC by hashing the index along with the least significant 48 bits of its virtual memory address. With this HMAC we can detect if the adversary tries to replace a code pointer with another pointer harvested from a different memory location. We find that SipHash [4], which is optimized for short messages, is a good choice of HMAC for our approach.

The middle third of Figure 5 illustrates the case where the function `foo` calls `bar` indirectly through a function pointer. Again, dark grey labels highlight our extensions to Readactor's code pointer hiding technique. The indirect call site in `foo` loads the (HMAC, index) pair from memory, recomputes the HMAC using the (address, index, key) tuple, and compares the two (see rightmost

third of Figure 5). If HMACs match, the index is used to lookup the address of the forward pointer which is subsequently used to execute control-flow edge ⑥. Notice that the forward trampoline that creates edge ⑦ does not target the first instruction in `bar`; instead, we add a delta to the address of `bar` to skip the forward cookie check that authenticates direct calls to `bar` (*e.g.*, edge ②).

As explained in Section 3.3, our attack against perfect execute-only memory swaps two pointers to hijack the program control flow. Because the address of the pointer is used to compute the HMAC, moving the pointer without re-computing the HMAC will cause the HMAC check before all indirect calls to fail unless the two (address, index) pairs collide in the hash. Attackers can still harvest and swap (HMAC, index) pairs stored to the same address at different times. See Section 8.1 for a more complete security analysis.

Returns from indirect calls make up the fourth and final class of control flows that we must authenticate. The callee sets a return cookie before the callee returns and check the cookie at the return site; see edges ⑧ and ⑨ in Figure 5. We again clear the cookie register directly after the check to prevent leaks. The cookie check at the end of arrow ⑨ must pass for all potential callees. Therefore, we set the lower 32-bits of all backward cookies to the same global random value and only check the lower halfword of the backward cookie at the return site. This ensures that returns only target return sites; however, any return instruction can target indirect call-preceded gadgets under this scheme. We did not reuse any indirect call-preceded gadgets in our harvesting attack since these are also protected by register randomization and callee-saved stack slot randomization. It is possible to further restrict returns from indirect calls by taking function types into account. Rather than setting the 32 lower bits of return cookies to the same random value, we can use different random values for different types of functions. We did not restrict backward control flows based on types in our prototype implementation of code pointer authentication since we would have had to manually find and whitelist any return sites following indirect calls to a type-incompatible callee.

7 Mitigating Implementation Challenges

As shown in Section 5, implementing a comprehensive execute-only memory protection policy is challenging. In contrast to our ICPOP attack, the FDMA and procfs attacks do not target conceptual but implementation weaknesses of execute-only memory defenses. Nevertheless, these issues must be addressed as they allow an attacker to completely bypass the deployed mitigation.

The FDMA and `/proc/self/mem` attack are the most imminent threat to execute-only defenses because they undermine the memory protection enforcement. The root cause for both attacks is that the DMA controller does not respect the set memory permissions. A straightforward approach is to use an IOMMU [3] because it was designed for exactly this purpose. However, not all platforms that support DMA feature an IOMMU; hence, we explored an alternative way to mitigate the FDMA attack. Our solution is based on the fact that an adversary cannot directly configure DMA controllers, since this requires kernel privileges, but relies on the kernel as a confused deputy. In general, the kernel already considers every input from the user mode as untrusted and checks pointers to memory against certain policies, but since current operating systems do not consider execute-only memory protection, read accesses to valid user memory do not violate any policy. We extend the policy check for user-mode pointers by iterating through meta-data structures of allocated memory regions and check whether they are allocated as execute-only memory. In particular, we apply our patch (38 LoC) to the `access_ok()` macro in the Linux kernel and check for the `VM_READ` permission in the `vm_area` data structure that corresponds to the address provided by the user mode before allowing read access. We further add a check that uses the patched `access_ok()` macro for addresses provided to the `/proc/self/mem` device before reading or writing the memory.

Providing a general defense against the other procfs attacks is challenging because it is baked into the Linux ecosystem as the needed native interface for many system utilities and programs. Hence, blocking access to it would disrupt a major kernel API and break a Linux distribution. The exposed nature of procfs has long been recognized and attacks proposed to exploit it especially with regard to differential privacy [34, 64]. While the impact on fine-grained code randomization, as deployed in Readactor, is limited because the granularity of the leaked information is coarser than the applied randomization[3], it can be exploited to bypass conventional ASLR. Although procfs cannot be removed entirely due to legitimate uses, some defenses have attempted to restrict access to procfs. Notably, GRSecurity's kernel patchset [56] has several configuration options to restrict access to `procfs` entries by user or group, with the intent that different critical processes can run as different users and be unable to compromise other processes. A recent defense [62] proposes falsifying information in procfs to mitigate other types of attacks. However, no defense prevents access by a program to its own procfs entry set, and any finer-grained procfs restriction by username would result in breaking benign applications.

[3] For example, `maps`, `numa_maps`, `smaps` can disclose the addresses of executable memory regions, however, fine-grained code randomization (*e.g.*, function permutation [35], basic block permutation [60], register randomization [47], etc.) hides the memory layout *within* this region.

8 Evaluation

8.1 Security

Code pointer authentication prevents reuse of the remaining exposed trampoline pointers, even if the attacker has harvested all available trampoline locations. Naturally, this authentication mitigates the attacks on ideal execute-only enforcement that we show in this work. To further demonstrate the security of code pointer authentication, we systematically consider each possibly exposed indirect branch target in turn.

Direct call trampoline entry (edge ① in Figure 5) An attacker can harvest the location of the backwards jump (`jump r_foo`) in the call trampoline from the return address on the stack. In the original Readactor defense, she can compute the address of the previous instruction from this pointer and invoke `t_bar`.

In direct call authentication, each direct callee function checks that its specific, per-function cookie is set prior to calling it. If the attacker cannot forge the callee function's cookie, this check will fail. We store the cookie as an immediate value in execute-only memory and pass it to the callee in a register. After performing the cookie check, the callee clears the register. Thus, direct call cookies cannot leak to an adversary, and the attacker has a 2^{-64} chance of successfully guessing the correct 64-bit random cookie value. Since the attacker cannot forge a correct cookie before an indirect branch to a direct call cookie, direct call trampoline entry points are unavailable as destinations for an attack.

Direct call trampoline return (edge ③ in Figure 5) Harvesting a return address corresponding to a direct call trampoline gives the attacker the location of the backwards jump in a call trampoline. In Readactor, this destination allows the attacker to invoke a call-preceded gadget beginning at `r_foo` in the example.

We also protect these destinations with an analogous, function-specific return cookie. Directly before a callee function returns, it sets its function-specific return cookie. The return site verifies that the expected callee's return cookie was set before continuing execution. As in the forward case, this prevents the attacker from reusing this destination, since she cannot forge a correct cookie.

Indirect call trampoline entry (edge ⑤ in Figure 5) Similarly, an attacker can harvest indirect call trampoline locations from the stack and dispatch to the beginning of an indirect call trampoline. However, this destination is trivial to the attacker, since she must set another valid, useful destination for the indirect call before invoking the trampoline. The attack could always dispatch straight to this final destination instead of the indirect call trampoline. Thus, we do not need to protect indirect call trampoline entry points from reuse.

Indirect call trampoline return (edge ⑧ in Figure 5) Analogous to the direct call case, the attacker can dispatch to the backwards edge of an indirect call trampoline to invoke an indirect-call proceeded gadget. This is a more challenging edge to protect without static analysis, since the indirect call site cannot know which function-specific return cookie to check.

Since the caller does not know the precise callee, we enforce a weaker authentication check on indirect call return destinations. By splitting return cookies into a global part and function-specific part, we can still ensure that the return site must be invoked by a return, not an indirect call.

We note that the fine-grained register randomization implemented in Readactor largely mitigates the threat of indirect-call proceeded gadget reuse, since the attacker cannot be sure of the semantics of the gadget due to execute-only memory.

Function trampolines (edge ⑥ in Figure 5) Function trampoline harvesting and reuse is the easiest attack vector against code-pointer hiding schemes. In Readactor, after harvesting function trampolines, the attacker can overwrite any return address or function pointer with a valid function trampoline destination and perform whole-function reuse.

We prevent reuse of function trampolines by changing the function pointer format to include an HMAC tying the function pointer to a specific memory address. This prevents reuse of function pointers from returns as well as swapping of function pointers in memory.

Since function pointers are no longer memory addresses in our authentication scheme, the attacker cannot use a function pointer as a return address at all. The return would interpret the address as an HMAC—Idx pair and fail to verify the HMAC, crashing the program.

Function pointers cannot be swapped arbitrarily under this defense, since the pointer is tied to its address in memory by the HMAC. If a pointer P at address A is moved to address B, the HMAC check when it loaded from address B will fail. Thus the attacker must either forge a valid HMAC or have harvested P from the targeted location in memory at a previous point in execution.

HMAC Forgery We first address the possibility of forging a valid HMAC for a function and pointer address pair without ever having seen a valid HMAC for that pair. SipHash is designed to be forgery-resistant, thus the probability of correctly forging a valid HMAC for a pointer at an address not previously HMACed is expected to be 2^{-48}, based on the size of the HMAC tag. Additionally, since we can store the HMAC key in execute-only memory, an attacker cannot disclose the 128-bit key, and thus is limited to brute-forcing this key.

Replay Attacks As in other pointer encryption schemes [17, 42], HMACs do not provide temporal safety against replay attacks on function pointers. That

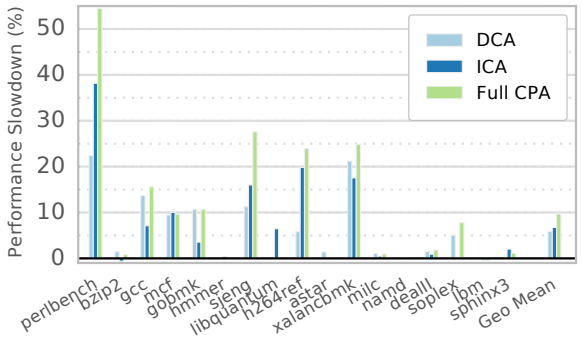

Figure 6: Performance overhead of code pointer authentication on SPEC CPU2006. All measurements include the overhead of the Readactor++ transformations.

is, a function pointer can be harvested at one point in program execution and later rewritten to the same address.

If we augmented the defense with the capability to track all function pointers in memory, we could re-key all HMACs at random times to prevent replay attacks. However, such instrumentation would require substantial bookkeeping overhead.

8.2 Performance

Code Pointer Authentication To evaluate the performance of our defensive techniques, we applied our improved protections on top of the Readactor++ system. We measured the performance overhead of both direct call authentication and function pointer authentication on the SPEC CPU2006 benchmark suite. These results are summarized in Figure 6. All benchmarks were measured on a system with two Intel Xeon E5-2660 processors clocked at 2 Ghz running Ubuntu 14.04.

With all protections enabled, we measured a geometric mean performance overhead of 9.7%. This overhead includes the overhead from basic Readactor call and jump trampolines and compares favorably with the 6.4% average overhead reported by Crane et al. [19]. We also measured the impact of direct call authentication and indirect call authentication individually (labeled DCA and ICA in the figure, respectively). We found that indirect code pointer authentication generally adds more overhead (6.7% average) than direct code pointer authentication (5.9% average), although this is strongly influenced by the program workload, specifically the percentage of calls using function pointers.

We observed that `h264ref` stands out as an interesting outlier for indirect call authentication. This benchmark repeatedly makes a call through a function pointer in a hot loop. To make matters worse, the target function is a one-line getter, thus our instrumentation dominates the time spent in the callee. This benchmark in particular benefits greatly from inlining the HMAC verification to avoid the extra call overhead. To speed up HMAC verification, especially in this edge case, we implemented a small (128 byte), direct-mapped, hidden cache of valid HMAC entries. This hidden cache is only accessed via offsets embedded in execute-only memory and is thus tamper-resistant. Before recomputing an HMAC, the verification routine checks the cache to see if the HMAC is present.

We found three corner cases in SPEC where we could not automatically compute a new HMAC when a function pointer was moved. This is because the program first casts away the function pointer type then copies the pointer inside a struct. We had to insert a single manual HMAC in `gcc` and another in `povray` to handle these edge cases. `perlbench` stores function pointers in a growable list, which is moved during reallocation. Since our prototype does not yet instrument the libc realloc function (although it could), we had to manually instrument these operations. The CCFI [42] HMAC scheme requires similar modifications. Finally, Readactor is not fully compatible with C++ exception handling, so we were not able to run `omnetpp` and `povray` which require exception handling.

DMA Mitigation Our patch to the kernel `access_ok()` macro has no measurable performance impact. Specifically, we measured no average performance overhead on the SPEC CPU2006 benchmark suite, as is expected since the benchmarks interact minimally with the kernel. To test real-world performance, we ran three performance measurements tools (`weighttp`, `httperf`, `httpress`) against the web server Nginx. Similar to the SPEC CPU2006 benchmarks we observed in some cases a performance improvement or degradation by a few hundredths of a percent which again is indistinguishable from measurement noise.

9 Generality of Attacks

The attacks described in this paper are generally applicable to many randomization-based defenses. Defenses that do not provide leakage resilience are generally vulnerable to various forms of information leakage attacks. Therefore, we focus on those that offer (some) resilience.

Direct leakage refers to attacks that read code pages, while indirect leakage refers to attacks that leak code addresses from stack or heap during execution. Since ICPOP attacks leak hidden or indirection (*e.g.*, trampoline) pointers indirectly from stack or heap, they are a form of indirect leakage attacks. Also, since non-TLB-mediated leakages directly read code pages (using mechanisms not protected by memory permissions), they are a form of direct information leakage. Accordingly, there are four sub-classes of information leakage: direct leakage via TLB-mediated code reads, direct leakage via non-TLB-mediated code reads, indirect leakage of code pointers, and indirect leakage of indirect code pointers.

Table 1: Defenses protecting against different classes of information leakage attacks

Defenses	Direct Leaks		Indirect Leaks	
	TLB-Mediated (*e.g.*, buffer over-read [57])	**Non-TLB-Mediated** (*e.g.*, DMA/ procfs § 5)	**Code Pointer Leaks** (*e.g.*, Ret addr. leak [21])	**Indirect Code Pointer Leaks** (*e.g.*, ICPOP § 3)
PointGuard [17]			✓	
Oxymoron [6]			✓	
Isomeron [21]			✓	
XnR [5]	✓			
HideM [27]	✓			
Readactor [19, 20]	✓		✓	
Heisenbyte [59]	✓			
ASLR-Guard [41]			✓	
TASR [8]	✓[4]	✓[4]	✓[4]	✓[4]
CPA § 6	✓	✓	✓	✓

Our attacks are applicable to randomization defenses regardless of the granularity or type of randomization. For example, various randomization defenses propose library-level [48], function-level [35], or instruction-level [31] randomization approaches. In ICPOP, we abuse and chain indirect code pointers to achieve control flow hijacking. Regardless of how the underlying code has been randomized, as long as the semantics remain intact, our profiling attack remain applicable. In attacks that use implementation challenges (FDMA or procfs), the exact contents of code pages are read (via non-TLB-mediated accesses), so regardless of the how intrusive the randomization is, we can disclose the randomized code and can then perform a traditional ROP attack.

Table 1 summarizes leakage-resilient randomization defenses and their vulnerabilities to various types of attacks. We briefly discuss each defense and how our attacks apply in the following.

PointGuard [17] protects all pointers stored in memory by masking them with an XOR key. It therefore prevents leakage of code addresses via pointers. However, indirect leakage of encrypted pointers and direct leakage attacks remain possible.

Oxymoron [6] attempts to prevent JIT-ROP attacks by adding a layer of indirection to instructions such as branches that reference other code pages. While Oxymoron thwarts the recursive disassembly step of the original JIT-ROP attack, it does not protect all pointers to code. Davi *et al.* [21] show an attack against Oxymoron, exploiting indirect address leakage. They then propose Isomeron that combines execution-path randomization with code randomization to build indirect leakage resistance. Neither of these techniques prevent direct code reads.

XnR [5] and HideM [27] perform permission checks on memory accesses to implement execute-only, thus preventing TLB-mediated code reads. They, however, do not check non-TLB-mediated code reads. They are also vulnerable to indirect leakage attacks, since code pointers are not hidden or protected in any way during execution. Leakage of return addresses or function pointers from the stack or heap remains possible during execution.

Readactor [19] utilizes Extended Page Table permissions to enforce execute-only permission and adds a layer of indirection (trampolines) to prevent indirect leaks. Therefore, it prevents TLB-mediated direct code reads and indirect leaks of code pointers (*e.g.*, return addresses and function pointers). Its permissions, however, do not apply to non-TLB-mediated accesses as demonstrated in Section 5. Moreover, leakage of trampoline pointers (*i.e.*, indirect code pointers) are possible as demonstrated by our ICPOP attack against Apache and Nginx.

Heisenbyte [59] prevents executable region leakages by making any code-area read destructive. It can only mitigate TLB-mediated direct leakages. Non-TLB-mediated memory accesses do not cause a byte destruction; thus, they are not mitigated. Indirect leakages also remain possible because code pointers are not protected in any way.

ASLR-Guard [41] provides leakage resistant ASLR by decoupling code and data, storing code locators in a secure region of memory, and encrypting code locators that are stored in observable memory. As a result, code locators themselves cannot leak because they are encrypted whenever placed in regular memory. However, the encrypted forward pointers can be profiled and reused by an attacker. This is hinted at in the paper itself: "... attackers may reused [*sic*] the leaked encrypted code locators to divert control flow." Direct code reads, whether they are through TLB (*e.g.*, buffer over-reads) or not, also remain possible in ASLR-Guard.

TASR [8] re-randomizes code regions at every I/O system call pair to mitigate any potential information leakage. It also fixes the code pointers on the stack and heap for every re-randomization. It can potentially mitigate all classes of remote leakage attacks, but it requires source code for compilation and it cannot mitigate leakages within the application boundary (*e.g.*, in JIT-ROP attacks).

[4] TASR is only applicable to ahead-of-time compiled code and mitigates exfiltration-style information leakage that cross the system call boundary. TASR is not applicable to leaks in interpreted code (*e.g.*, scripts in browser) that do not cross the system call boundary, such as the JIT-ROP [55] attack.

10 Related Work

Our work mainly relates to memory corruption attacks and defenses. The literature in the areas of software diversity, control-flow integrity, and code pointer protection is vast. We refer the interested reader to the relevant surveys [38, 58] and focus on closely related work.

Automatic Software Diversity Address Space Layout Randomization (ASLR) is the most widely used randomization-based defense [10, 48]. However, ASLR provides insufficient entropy on 32-bit systems [54] and brute force attacks are sometimes possible even against 64-bit systems [9]. ASLR can also be bypassed via information leakage attacks [16, 52, 57]. This motivated approaches that randomize the code at a finer-granularity, *e.g.* at the level of functions [30, 35], basic blocks [23, 60], or instructions [31, 32, 47]. The assumption that finer-grained diversity addresses the shortcomings of ASLR was undermined by JIT-ROP [55] and side-channel attacks [7, 33, 51] that directly or indirectly disclose the randomized code layout. This motivated work on leakage-resilient code randomization. The Oxymoron defense [6] prevents the recursive disassembly while later approaches more generally prevent read accesses to code pages [5, 12, 19, 27]. An alternative strand of research explores techniques that tolerate leakage by periodically re-randomizing the code layout [8, 28, 40], randomizing or restricting the control-flow [21, 43], or by implementing destructive reads [59].

Code Pointer Protection Directly reading the code is not the only way adversaries can disclose the code layout. Davi et al. demonstrated that virtual method tables can be used to indirectly disclose enough code to mount a JIT-ROP attack [21]. PointGuard [17] was the first defense to protect pointers by XOR'ing them with a secret key. PointGuard, however, is not secure in our threat model which assumes that adversaries can read and write arbitrary memory. Readactor [19] introduced code pointer hiding in which decouples the function layout from pointers in attacker-observable memory thanks to X-only memory. ASLR-Guard [41] uses the vestiges of x86 segmentation to encrypt code pointers (code locators) in attacker-observable memory. As mentioned in our analysis of leakage-resilient defenses (Section 9), code locators can still be profiled and used for ICPOP attacks.

General memory safety approaches such as Softbound [44] and CETS [45] prevent corruption of code pointers but incur high performance overheads. The recent code-pointer integrity (CPI) technique demonstrates that providing memory safety for code pointers by isolating them in a safe region is more efficient than providing complete memory safety [37]. However, it was later discovered that information hiding is not a safe way to isolate code pointers even in a 64-bit address space [24]. In contrast to CPI, cryptographically-enforced CFI (CCFI) protects code pointers without isolating them in memory [42]. Similar to our CPA approach, CCFI [42] computes HMACs for code pointers. CCFI protects all pointers with cryptographically-secure HMACs generated using 128-bit AES encryption. We do not use AES encryption to generate HMACs due to its high overhead; the authors of CCFI report an average SPEC-CPU2006 overhead of 52% and worst case slowdowns of a factor of 2.7x. Since we layer CPA on top of the protection offered by execute-only memory and code pointer hiding, we need only to protect forward (indirect) code pointers. Additionally we can store the HMAC inside a single word-sized pointer, since we need only to store a trampoline index rather than a function's memory address. Finally, we chose to use SipHash [4] rather than a more expensive AES HMAC which further decreases the cost of our technique.

Other Related Work CFI validates the control flow at each indirect branch instruction [1]. Most CFI implementations compute an approximate control-flow graph (CFG) in an offline step and use it to validate indirect branch targets at runtime. Hence, CFI is not directly affected by our attacks and is in principle immune against information disclosure [2]. However, implementing CFI in a production compiler is very challenging and seemingly small errors or coverage gaps can allow adversaries to avoid enforcement [39]. The granularity of CFI policies is an even bigger challenge. Efficient, coarse-grained CFI approaches [63, 65] allow so many superfluous edges that adversaries still construct so called gadget-stitching attacks that adhere to the enforcement policy [14, 22, 29]. The inherent imprecision in static program analysis means that even fine-grained CFI policies are not always sufficient to prevent code-reuse attacks against applications with large code bases [13, 25].

The recent COOP technique also questions the security of binary-only CFI schemes as they lack type and class hierarchy information for C++ programs which is necessary to enforce fine-grained CFI [50]. Our CPP technique demonstrates that the COOP technique even applies to procedural languages.

11 Conclusion

In this paper, we evaluated the effectiveness of leakage-resilient code randomization. We presented two generic classes of attacks that can bypass ideal and practical execute-only defenses including the state-of-the art scheme, Readactor, and built three realistic exploits. Moreover, we proposed, implemented, and evaluated a new defense called code pointer authentication. Our mitigation adds little additional overhead. Our findings show that preventing information leakage is surprisingly hard because execution alone leaks valuable information.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security* (2005), CCS.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. A theory of secure control flow. In *7th International Conference on Formal Methods and Software Engineering* (2005), ICFEM'05.

[3] AMD, I. I/O virtualization technology (IOMMU) specification. *AMD Pub 34434* (2007).

[4] AUMASSON, J.-P., AND BERNSTEIN, D. J. SipHash: A fast short-input PRF. In *13th International Conference on Cryptology in India* (2012), INDOCRYPT.

[5] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS.

[6] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium* (2014), USENIX Sec.

[7] BARRESI, A., RAZAVI, K., PAYER, M., AND GROSS, T. R. CAIN: Silently breaking aslr in the cloud. In *9th USENIX Security Symposium* (2015), WOOT'15.

[8] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. Timely rerandomization for mitigating memory disclosures. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[9] BITTAU, A., BELAY, A., MASHTIZADEH, A. J., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *35th IEEE Symposium on Security and Privacy* (2014), S&P.

[10] BOJINOV, H., BONEH, D., CANNINGS, R., AND MALCHEV, I. Address space randomization for mobile devices. In *ACM Conference on Wireless Network Security* (2011), WiSec.

[11] BOWDEN, T., BAUER, B., NERIN, J., FENG, S., AND SEIBOLD, S. The /proc filesystem, 2009.

[12] BRADEN, K., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., LIEBCHEN, C., AND SADEGHI, A.-R. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium* (2016), NDSS.

[13] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium* (2015), USENIX Sec.

[14] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium* (2014), USENIX Sec.

[15] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *ACM SIGSAC Conference on Computer and Communications Security* (2010), CCS.

[16] CHEN, X. ASLR bypass apocalypse in recent zero-day exploits, 2013.

[17] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium* (2003), USENIX Sec.

[18] CRANE, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Booby trapping software. In *New Security Paradigms Workshop* (2013), NSPW.

[19] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy* (2015), S&P.

[20] CRANE, S., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., SUTTER, B. D., AND FRANZ, M. It's a TRaP: Table randomization and protection against function-reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[21] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium* (2015), NDSS.

[22] DAVI, L., SADEGHI, A., LEHMANN, D., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium* (2014), USENIX Sec.

[23] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., AND SADEGHI, A. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security* (2013), ASIACCS.

[24] EVANS, I., FINGERET, S., GONZALEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy* (2015), S&P.

[25] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[26] FAULKNER, R., AND GOMES, R. The process file system and process model in unix system v. In *USENIX Technical Conference* (1991), ATC.

[27] GIONTA, J., ENCK, W., AND NING, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy* (2015), CODASPY.

[28] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium* (2012), USENIX Sec.

[29] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *35th IEEE Symposium on Security and Privacy* (2014), S&P.

[30] GUPTA, A., KERR, S., KIRKPATRICK, M., AND BERTINO, E. Marlin: A fine grained randomization approach to defend against ROP attacks. In *Network and System Security*, Lecture Notes in Computer Science. 2013.

[31] HISER, J., NGUYEN, A., CO, M., HALL, M., AND DAVIDSON, J. ILR: Where'd my gadgets go. In *33rd IEEE Symposium on Security and Privacy* (2012), S&P.

[32] HOMESCU, A., JACKSON, T., CRANE, S., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Large-scale automated software diversity—program evolution redux. *IEEE Transactions on Dependable and Secure Computing PP*, 99 (1 2015), 1. Pre-Print.

[33] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symposium on Security and Privacy* (2013), S&P.

[34] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *33rd IEEE Symposium on Security and Privacy* (2012), S&P.

[35] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference* (2006), ACSAC.

[36] KILLIAN, T. J. Processes as files. In *USENIX Association Software Tools Users Group Summer Conference* (1984), STUG.

[37] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI.

[38] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy* (2014), S&P.

[39] LIEBCHEN, C., NEGRO, M., LARSEN, P., DAVI, L., SADEGHI, A.-R., CRANE, S., QUNAIBIT, M., FRANZ, M., AND CONTI, M. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[40] LU, K., NÜRNBERGER, S., BACKES, M., AND LEE, W. How to make aslr win the clone wars: Runtime re-randomization. In *23rd Annual Network and Distributed System Security Symposium* (2016), NDSS.

[41] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[42] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. CCFI: cryptographically enforced control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[43] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K., AND FRANZ, M. Opaque control-flow integrity. In *Annual Network and Distributed System Security Symposium* (2015), NDSS.

[44] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), PLDI.

[45] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management* (2010), ISMM.

[46] OPENBSD. Openbsd 3.3, 2003.

[47] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy* (2012), S&P.

[48] PAX. PaX address space layout randomization, 2003.

[49] SANG, F. L., NICOMETTE, V., AND DESWARTE, Y. I/O attacks in Intel PC-based architectures and countermeasures. In *SysSec Workshop* (2011), SysSec.

[50] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy* (2015), S&P.

[51] SEIBERT, J., OKHRAVI, H., AND SÖDERSTRÖM, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS.

[52] SERNA, F. J. cve-2012-0769, the case of the perfect info leak, 2012.

[53] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security* (2007), CCS.

[54] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proc. of ACM CCS* (2004), pp. 298–307.

[55] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy* (2013), S&P.

[56] SPENGLER, B. Grsecurity. *Internet [Nov, 2015]. Available on: http://grsecurity.net* (2015).

[57] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security* (2009), EUROSEC.

[58] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Proc. of IEEE Symposium on Security and Privacy* (2013).

[59] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[60] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security* (2012), CCS.

[61] WOJTCZUK, R. Subverting the xen hypervisor. In *Blackhat USA* (2008), BH US.

[62] XIAO, Q., REITER, M. K., AND ZHANG, Y. Mitigating storage side channels using statistical privacy mechanisms. In *ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS.

[63] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy* (2013), S&P.

[64] ZHANG, K., AND WANG, X. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *18th USENIX Security Symposium* (2009), USENIX Sec.

[65] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium* (2013), USENIX Sec.

Appendix A: Nginx Attack 1 Details

11.0.1 Locating a mutex

During execution, Nginx makes several `open` system calls. During these calls, the address of the trampoline (*i.e.*, indirect code pointer) for `open` is vulnerable to being read by an attacker. However, in practice, determining the exact address of this trampoline is difficult. Furthermore, the attacker would have to perform a read within the very narrow window of opportunity in which the address is on the stack. We overcome this difficult by employing MTB. glibc's threading implementation supports a feature known as *thread cancellation*. There are two forms of cancellation: *asynchronous*, which means a thread's execution can be cancelled at any point in its execution and *deferred* which means any cancellation requests are deferred until a special predetermined point known as a cancellation point.

Every thread in a program contains a Thread Control Block (TCB). This structure contains thread-specific information and is used by glibc for keeping track of things such as *thread local storage*, the current extent of the thread's stack, and thread cancellation. Inside the TCB is a field named `cancelhandling`. This field contains flags representing various aspects of a thread's cancellation state. We are concerned with the following flags:

- TCB_CANCELTYPE: can this thread be cancelled asynchronously via a signal?
- TCB_CANCELING: is the thread's cancellation state being mutated?
- TCB_CANCELED: has the thread successfully canceled?

Before entering a cancellation point, glibc executes __pthread_enable_asynccancel, a function that enables asynchronous cancellation by setting TCB_CANCELTYPE to true. After exiting a cancellation point, glibc executes __pthread_disable_asynccancel, a function that disables asynchronous cancellation by setting TCB_CANCELTYPE to false. A thread's cancellation can be requested by calling pthread_cancel which will set TCB_CANCELED to true if asynchronous cancellation is disabled. If asynchronous cancellation is enabled the requesting thread will send a signal to the target thread and a signal handler will mark the thread for cancellation. This use of signals creates the possibility of a data race: if a thread is in the process of requesting a cancellation and the target thread disables asynchronous cancellation before the requesting thread sends its signal, the target thread will be forced to execute its cancellation handler while in an unexpected state. To prevent this, before sending a signal, the requesting thread uses a *Compare and Exchange* instruction that can ensure TCB_CANCELING is false, TCB_CANCELTYPE is true and set TCB_CANCELING to true atomically. pthread_cancel performs this instruction in a loop until it succeeds.

Analogously, upon exiting a cancellation point, a thread uses a *Compare and Exchange* instruction to both ensure TCB_CANCELING is false and to set TCB_CANCELTYPE to false. This instruction is also executed in a loop until it succeeds. This makes TCB_CANCELING a Mutual Exclusion Device (mutex) that prevents concurrently disabling asynchronous cancellation and sending an asynchronous cancellation signal. By setting TCB_CANCELING to true, an attacker can force a thread to loop in __pthread_disable_asynccancel, forever waiting for a signal that will never come.

Many cancellation points map directly to system calls and these system calls are surrounded by __pthread_enable_asynccancel and __pthread_disable_asynccancel. A simplified example of glibc's implementation of open is presented below:

```
__pthread_enable_asynccancel();
open syscall;
__pthread_disable_asynccancel();
```

Since glibc's open function uses a cancellable system call we can profile a trampoline for open by setting TCB_CANCELING to true and reading it off the stack when it hangs in the __pthread_disable_asynccancel after open.

Having identified a suitable mutex for MTB, we then determine a way to locate it at runtime. As cancelhandling is a field of a thread's TCB, given the base address of a TCB, it is trivial to locate cancelhandling. In glibc every TCB contains a header with the type tcbhead_t. The first field of this structure is defined as void *tcb; which is, actually, just a pointer to itself. The fact that the TCB begins with a pointer to itself makes it easily distinguishable in memory. Once we locate a thread's TCB we can leverage cancelhandling to execute MTB against the thread. Additionally, since all TCBs are connected via linked list pointers, locating a single TCB allows us to locate the TCBs of all other threads.

11.0.2 Profiling open

Using the mutex found in the previous section, we can cause a thread of our choosing to hang at a nondeterminate system call. By modifying TCB_CANCELING to false then true in quick succession, we can permit that thread's execution to continue and then stop again at a nondeterminate system call. As Nginx makes many system calls which involve cancellation points, locating open requires the ability to distinguish when a thread is hung at open versus when it is hung at some other cancellable system call. We distinguish these situations by exploiting knowledge of how Nginx responds to requests for static files.

When Nginx receives an HTTP GET request for static content, it transforms the requested path into a path on the local filesystem. It calls open on this path and, if successful, responds with the file's contents. If the open fails, it responds with HTTP 404 Not Found. During this process, a pointer to a string containing the path will be present on the stack. To determine whether or not Nginx is hung at an open call we craft an HTTP request with a unique string and examine all strings pointed to from Nginx's stack. An example request is provided below.

```
GET /4a7ed3b71413902422846 HTTP/1.1
```

11.0.3 Profiling _IO_new_file_overflow

We profile _IO_new_file_overflow by taking advantage of glibc's implementation of the stdio FILE type. Every FILE contains a file descriptor, pointers to the file's buffers, and a table of function pointers to various file operations. _IO_new_file_overflow is included among these function pointers. By locating a valid FILE, we can easily locate _IO_new_file_overflow as the ordering of functions within the table is fixed. Finding a valid FILE pointer in Nginx proved to be a challenge as Nginx uses file descriptors instead of FILE. In this situation, scanning the stack will not yield a pointer to a valid FILE. To overcome this, we locate glibc's FILE for the standard output stream stdout. stdout is a global variable and is always automatically initialized on startup.

Since stdout is a global variable defined by glibc, it is located in glibc's data segment. Due to ASLR, the location of glibc's data segment cannot be known *a priori*; nor can it be directly inferred from the address of the stack. Attack is further complicated by the fact that we cannot dereference random stack values due to the risk of causing a segmentation fault.

Instead, we find a pointer into the heap, which occur more frequently in the stack. While pointers into the heap are common, they are not easily distinguishable from non-pointer values. To distinguish heap pointers we perform a simple statistical analysis on the values of the stack, the details of which we will present in a Technical Report for the sake of brevity. We found that, for Nginx, 2 pages of values collected at a single point in time is enough to reliably distinguish heap pointers.

Now that we have pointers into the heap, it becomes possible for us to analyze the heap. We leverage this to find a pointer to main_arena, a glibc global variable. main_arena is a structure used by glibc to maintain information on allocated chunks of memory. To speed up allocation operations, glibc partitions chunks into presized bins and stores them in main_arena. Every heap chunk allocated via malloc, calloc, or realloc is prefixed with metadata containing a pointer back to the main_arena bin it came from. We take advantage of this do locate a pointer into main_arena.

Starting from the smallest pointer in our bin of heap pointers, we collect 8-byte values from a 20 page range of the heap. We then filter out values unlikely to be pointers.

Finally, we partition the remaining values into bins of size 0x100000. The most common pointer of the largest bin will be a pointer into main_arena. This is due to most chunks of the heap being allocated out of the same bin.

Now that we have a pointer into glibc's data section we can search for stdout. We identify stdout by scanning backwards from main_arena, and looking for a region that is both a valid FILE and has the value 1 for its underlying file descriptor. At this point, the location of _IO_new_file_overflow can be trivially read off of stdout.

11.0.4 Corrupting the Nginx task queue

The main loop for Nginx worker threads is located in ngx_thread_pool_cycle. All new worker threads spin in this loop, checking if new tasks have been added to their work queue. A simplified version of this loop is presented below:

```
for (;;) {
    task = queue_get(tp->task_queue);
    task->handler(task->ctx, tp->log);
}
```

To carry out our attack, we craft a fake task structure. We construct our fake task in the region of the stack that originally contained Nginx's environment variables. At startup Nginx copies these to a new location and the original location goes unused.

We initialize our fake task such that task->handler points to open and task->ctx points to html/index.html. We also modify tp->log to be equal to (O_DIRECT | O_SYNC | O_WRONLY | O_TRUNC) . While this invalidates the tp->log pointer, in practice, threads do not seem to log unless Nginx is compiled in debug mode. When the worker thread goes to execute this task, it will open the file in O_DIRECT mode, allowing us to perform an FDMA attack.

Once we have our counterfeit task structure, we can append it to the task queue and wait for Nginx to execute the task. This usually happens instantaneously, so after a few seconds we can be confident our call has occurred. We repeat this process 100 times so that there will be at least 100 file descriptors in O_DIRECT mode opened by the Nginx process.

For the call to _IO_new_file_overflow, we begin by creating a fake FILE that matches stdout except for the following fields:

1. file->file_fileno = 75
2. file->file_IO_write_base = file->vtable->__overflow & 0xFFF
3. file->file_IO_write_ptr = file->file_IO_write_base + 0x1000
4. file->file_IO_read_end = file->file_IO_write_base

Next, we modify our fake task such that task->handler points to _IO_new_file_overflow and task->ctx points to our fake FILE. We also modify tp->log to be -1 EOF. This will cause _IO_new_file_overflow to think the write buffer overflowed just as the end of the file was reached, so it will immediately flush the buffer via a write. Once we have crafted our fake arguments we append the fake task to the task queue and wait for the task to be executed. Conceptually _IO_new_file_overflow will be executing the equivalent of the following code:

```
write(75, \_IO\_FILE\_Overflow & ~0xFFF, 0x1000);
```

Which results in an FDMA from execute only memory into the file html/index.html. We can then retrieve this page of code by sending GET /index.html HTTP/1.1. We now have the contents of a page of code at a known location and can proceed with a standard ROP attack. If necessary, we can perform this as many times as we want to leak more pages of memory.

Appendix B: Nginx Attack 2 Details

Nginx's design employs a master process, which provides signal handling and spawns worker processes to handle requests via fork calls. This processing loop is implemented by the ngx_master_process_cycle

```
ngx_argv[0] = "/usr/bin/python3"
ngx_argv[1] = "-c"
ngx_argv[2] = "import os,socket,subprocess;
   s=socket.socket(socket.AF_INET,
     socket.SOCK_STREAM);
   s.connect((\\\'127.0.0.1\\\',1234));
   [os.dup2(s.fileno(),i) for i in range(3)];
   subprocess.call([\\\'/bin/sh\\\',\\\'-i\\\']);"
 ngx_argv[3] = 0
```

Figure 7: Reverse Shell in Nginx with ICPOP

function, which is called from `main` after Nginx configures itself. The trampoline address of this function can be determined via profiling after causing a system call to hang. Since `ngx_master_process_cycle` forks, worker processes inherit the parent's current stack. This includes the return address trampoline of `ngx_master_process_cycle`. Recall that return addresses are replaced with a pointer to a trampoline whose code resembles the following:

```
call  ngx_master_process_cycle
jmp callsite_main
```

The return address points to the `jmp` instruction. From that address, we can easily derive where the `call` instruction is.

Identifying the relevant return address on the stack is straightforward, as Nginx's initial execution is predictable. The `ngx_master_process_cycle` frame will be near the base of the stack, immediately after the `main` stack frame.

Once the address of `ngx_master_process_cycle` is found, we can take advantage of a function pointer in the Nginx worker's log handler. The `log_error_core` function contains a pointer to a log handler function taking three arguments: `p = log -> handler(log, p, last-p)`. There are multiple system calls in the function prior to the pointer being dereferenced during a logging event, which enables us to hang the program via mutex-based MTB and corrupt the handler to point instead at `ngx_master_process_cycle`. In order to prevent a program crash, we must also modify the first argument (`log`) to resemble the `ngx_cycle_t` expected by `ngx_master_process_cycle`. The parameter is not used in our attack, so any non-crashing value suffices.

Once we have pointed the log handler at `ngx_master_process_cycle`, we must ensure that the target function's execution causes an `exec` under our control. This can be achieved via the range of signals that Nginx can handle in `ngx_master_process_cycle`. In particular, Nginx provides a `new_binary` signal used to provide rolling updates to a new version of the server without compromising availability. This signal handler is invoked whenever a global integer variable named `ngx_change_binary` is non-zero. The path to the binary is stored in `ngx_argv`, another global variable. By corrupting the first global value we ensure that an `exec` call will eventually be made when the log handler pointer is dereferenced. By corrupting the latter, we ensure that a binary of our choice is executed. For example, setting `ngx_argv` to the values shown in Figure 7 will create a reverse shell bound to a chosen IP address (127.0.0.1 in this case).

Appendix C: Apache Attack Details

In order to maintain portability across operating systems Apache uses its own portable runtime libraries (APR and APR-Util) instead of directly calling functions in libc. However, modules may call functions in this library that the base Apache process does not. Each of these files contains function pointers to every function in that library. They are linked to the executable during program compilation, and loaded into the data section of memory on execution.

One of these exported functions is `ap_get_exec_line` in Apache's server utility library (`httpd.h`), which takes three arguments: a pointer to a valid memory pool, a command to run, and the arguments to supply that command. We recover the trampoline for this function by profiling while hanging execution via mutex-based MTB. The region of memory containing pointers from `exports.c` is easily identified, as it contains nothing but function pointers (with common higher-order bits) pointing to functions in one library. The order in which function pointers are declared in `exports.c` is deterministic, so recovering the pointer for `ap_get_exec_line` is straightforward.

Next, we corrupt a function pointer to point to the revealed address. When choosing the pointer, we must ensure that the parameters passed to `ap_get_exec_line` are passed correctly, as this attack does not rely on global variables like the Nginx variant. Additionally, our ability to modify memory is limited to the periods surrounding system calls. Only functions which pass parameters via pointers to memory addresses are viable. Given these criteria we chose to corrupt the `errfn` pointer in `sed_reset_eval`, part of Apache's `mod_sed`. The `errfn` pointer is dereferenced in the `eval_errf` function, which pulls all of its parameters from pointers to memory. Similar functions are available in other modules, should `mod_sed` not be available.

Finally, we set `errfn` to point to `ap_get_exec_line`. The first argument pointer is corrupted to point at a valid `apr_pool_t` object, which the attacker-controller worker will likely already have. (APR pools are used to handle memory allocation in Apache.) The second pointer is made to point at a string containing the path to a binary of our choice. When the `errfn` pointer is dereferenced, the binary is executed.